Liber Amicorum 97-115
En Prof. L. Bryt ed. 1996

Enkele actuele onderzoeksthema's in Computationele Logica

Maurice Bruynooghe* Bart Demoen† Danny De Schreye‡

Samenvatting

Computationele logica is een paradigma binnen de informatica en de kunstmatige intelligentie dat de nadruk legt op de declaratieve aspecten van de voor te stellen kennis en de te ontwikkelen programma's. Dit artikel schetst de grondbeginselen van het programmeren op basis van logica en gaat daarna in op de principes die de basis vormen van een drietal onderzoeksdomeinen binnen de computationele logica.

1 Inleiding

Computationele Logica, ook vaak *Logisch Programmeren* genoemd, is gegroeid uit het inzicht dat eenvoudige eerste-orde predicatenlogica de basis kan vormen voor praktische programmeertalen. Meer bepaald zijn deze talen gebaseerd op een uitbreiding van de logica der Horn-uitdrukkingen. Horn-uitdrukkingen nemen de volgende algemene vorm aan:

$$p(t_1, \ldots, t_n) : - \; q_1(s_{11}, \ldots, s_{1m_1}), q_2(s_{21}, \ldots, s_{2m_2}), \ldots, q_k(s_{k1}, \ldots, s_{km_k}).$$

Hierin zijn $p, q_1, q_2, \ldots, q_k$ *relatie-* of *predicaat-* symbolen, $t_1, \ldots, t_n$, $s_{i1}, \ldots, s_{im_i}$, $i = 1, k$, *termen*, en $p(t_1, \ldots, t_n), q_1(s_{11}, \ldots, s_{1m_1}), \ldots, q_k(s_{k1}, \ldots, s_{km_k})$ *atomen*. Termen zijn syntactische constructies die zijn opgebouwd met behulp van veranderlijken, constanten en functie-symbolen. Ze representeren de objecten in het beschouwde toepassingsdomein.

Enkele concrete voorbeelden van Horn-uitdrukkingen zijn:

```
vader(X, Y) :- ouder(X, Y), man(X).
inboom(Knoop, boom(Linkerboom, Waarde, Rechterboom)) :-
        inboom(Knoop, Linkerboom).
append(Eerste.Rest, Lijst, Eerste.Nieuwe_rest) :-
        append(Rest, Lijst, Nieuwe_rest).
horizontaal(segment(punt(X1,Y), punt(X2,Y))).
```

*K. U. Leuven, Departement Computerwetenschappen, Celestijnenlaan 200A, B-3001 Heverlee. Gesteund door het NFWO. Email: Maurice.Bruynooghe@cs.kuleuven.ac.be

†K. U. Leuven, Departement Computerwetenschappen, Celestijnenlaan 200A, B-3001 Heverlee. Email:Bart.Demoen@cs.kuleuven.ac.be

‡K. U. Leuven, Departement Computerwetenschappen, Celestijnenlaan 200A, B-3001 Heverlee. Gesteund door het NFWO. Email: Danny.DeSchreye@cs.kuleuven.ac.be

De bedoelde betekenis van deze formules is respectievelijk :

- Voor alle **X** en **Y** geldt: **X** is vader van **Y** als **X** ouder is van **Y** en **X** een man is.
- Voor alle **Knoop**, **Linkerboom**, **Waarde** en **Rechterboom** geldt: **Knoop** komt voor als een knoop in de binaire boom met als linkerdeelboom **Linkerboom**, als rechterdeelboom **Rechterboom** en als wortel **Waarde** als **Knoop** voorkomt als knoop in **Linkerboom**.
- Voor alle **Eerste**, **Rest**, **Lijst** en **Nieuwe_rest** geldt: de lijst met als eerste element **Eerste** en als rest de lijst **Nieuwe_rest** is de samenvoeging van de lijst met als eerste element **Eerste** en als rest de lijst **Rest** en de lijst **Lijst** als de lijst **Nieuwe_rest** de samenvoeging is van de lijst **Rest** en de lijst **Lijst**.
- Voor alle **X1**, **X2** en **Y** geldt: het lijn-segment door de punten met coördinaten (**X1** , **Y**) en (**X2** , **Y**) is horizontaal.

De eerste drie voorbeelden zijn logische implicaties: de uitdrukking in het linkergedeelte van de formule is de conclusie van de implicatie, de uitdrukking in het rechtergedeelte is de conditie, terwijl het symbool ":-" staat voor logische implicatie (" $\leftarrow$ "). De conclusie is steeds een *atomaire* relationele uitspraak. De conditie is een conjunctie van atomaire relationele uitspraken, gescheiden door het ","-symbool, dat conjunctie (" $\wedge$ ") uitdrukt. Als een bijzonder geval, geïllustreerd in het vierde voorbeeld, kan de conditie leeg zijn. De conclusie geldt dan onafhankelijk van enige conditie.

Alle woorden die beginnen met een hoofdletter stellen veranderlijken voor. Deze veranderlijken zijn steeds *universeel gekwantificeerd* ("$\forall$") over het geheel van de formule. De symbolen "boom" en " $\cdot$ " zijn functiesymbolen. Beide hebben ariteit twee. De functie " $\cdot$ " werd in infix-notatie geschreven.

Als uitbreiding op het formalisme van de Horn-uitdrukkingen worden normaal ook negaties van atomaire relationele uitspraken in het conditie-gedeelte van de implicaties toegelaten. Dit geeft dan regels zoals bijvoorbeeld:

```
broer(X, Y) :- man(X), ouder(Z, X), ouder(Z, Y),
               not gelijk(X, Y).
```

Een programma in een logische programmeertaal bestaat uit een sequentie van dergelijke (mogelijk met negatie uitgebreide) Horn-uitdrukkingen. Deze sequentie drukt een conjunctief verband uit: het programma is de conjunctie van de opgesomde uitdrukkingen.

Zo'n conjunctie van Horn-uitdrukkingen drukt uit hoe de atomaire relaties in de conclusiegedeeltes kunnen worden gedefinieerd in termen van de (meestal meer primitieve) atomaire relaties in de bijhorende conditiegedeeltes. In sommige gevallen volstaat één enkele uitdrukking om een dergelijke definitie uit te drukken. Dit is bijvoorbeeld het geval voor de bovenstaande relaties *vader*, *horizontaal* en *broer*. In andere gevallen zijn meerdere uitdrukkingen nodig teneinde een nieuwe relatie te definiëren. Zo drukken de bovenstaande voorbeelden voor *inboom* en *append* slechts een gedeelte van de definitie van de bedoelde concepten uit. In het geval van *append* is een tweede Horn-uitdrukking

```
append(nil, Lijst, Lijst).
```

nodig om de definitie te vervolledigen. Voor *inboom* zijn twee bijkomende uitdrukkingen (één voor het geval dat **Knoop** de wortel is van de boom, en één voor het geval dat **Knoop** in de rechterdeelboom voorkomt) nodig.

Tenslotte dienen we ook in staat te zijn om programma's op te roepen. Dit gebeurt met behulp van een *vraagstelling*.

Voorbeelden van vraagstellingen zijn:

```
?- append(1.nil, 2.3.nil, X).
?- append(X, Y, a.b.nil).
?- vader(X, jan), broer(X, ria).
```

De bedoelde betekenis van deze vraagstellingen is respectievelijk:

- bestaat er een lijst X, zodanig dat X de samenvoeging is van de lijsten **1.nil** en **2.3.nil**? Zo ja, geef (al dergelijke) X.
- bestaan er lijsten X en Y, zodanig dat de lijst **a.b.nil** de samenvoeging is van de lijsten X en Y? Zo ja, geef (al dergelijke) X en Y.
- bestaat er een X , zodanig dat X vader is van **jan** *en* X broer is van **ria**? Zo ja, geef (al dergelijke) X.

Indien we de vermelde Horn-uitdrukkingen aanvullen met:

```
ouder(piet, jan).
ouder(an, piet).
ouder(an, ria).
man(piet).
man(jan).
gelijk(X, X).
```

dan bekomen we een programma dat de antwoorden op al deze vraagstellingen ondubbelzinnig vastlegt. De concrete antwoorden zijn dan respectievelijk:

```
ja, X is 1.2.3.nil.
ja, X is nil en Y is a.b.nil, of X is a.nil en Y is b.nil,
     of X is a.b.nil en Y is nil.
ja, X is piet.
```

Wat tot nu toe nog niet werd besproken is de manier waarop dergelijk antwoorden uit het programma en de vraagstellingen worden berekend. Vooraleer we hier zeer bondig op ingaan is het belangrijk om op te merken dat we in het voorgaande de betekenis van programma's en wat ze als resultaten berekenen, hebben verduidelijkt zonder *enige* verwijzing naar het uitvoeringsmodel, en zelfs zonder enige aanduiding van de mogelijke procedurale interpretatie van de taalconstructies. Logische programmeertalen zijn declaratieve talen:

programma's kunnen begrepen worden onafhankelijk van het uitvoeringsmodel. Een standaardwerk waarin deze declaratieve semantiek toegelicht wordt is [4].

Het gebruikte uitvoeringsmechanisme steunt op de procedurale interpretatie van de Horn-uitdrukkingen, zoals die door Kowalski ontwikkeld werd en die hij uitvoerig toelicht in zijn boek [3]. Daarbij worden de atomen in de vraagstelling beschouwd als oproepen van procedures die gedefinieerd zijn door de Horn-uitdrukkingen die het betreffende predikaat in hun linkerdeel hebben. Een elementaire rekenstap bestaat uit een zogenaamde *resolutiestap*. We illustreren dit met een voorbeeld. Stel dat we als vraagstelling

```
?- append(1.nil, 2.3.nil, X), p(X).
```

hebben en dat de Horn-uitdrukking

```
append(Eerste.Rest, Lijst, Eerste.Nieuwe_rest) :-
        append(Rest,Lijst,Nieuwe_rest).
```

gebruikt wordt om het eerste atoom in de vraagstelling op te lossen. Resolutie begint met *unificatie* tussen het geselekteerde atoom in de vraagstelling en het linkerdeel van de geselekteerde Horn-uitdrukking. Daarbij wordt op de veranderlijken in die twee atomen een minimale subtitutie (most general unifier) toegepast zodat de twee atomen aan elkaar gelijk worden. Die substitutie toepassen op de vraagstelling en de Horn-uitdrukking geeft:

```
?- append(1.nil, 2.3.nil, 1.Nieuw_rest), p(1.Nieuwe_rest).
append(1.nil, 2.3.nil, 1.Nieuwe_rest) :-
        append(nil, 2.3.nil, Nieuwe_rest).
```

Vervolgens wordt een nieuwe vraagstelling afgeleid door het geselekteerde atoom in de geïnstantieerde vraagstelling te vervangen door het rechterdeel van de geïnstantieerde Horn-uitdrukking. Dit geeft

```
?- append(nil, 2.3.nil, Nieuwe_rest), p(1.Nieuwe_rest).
```

als nieuwe vraagstelling.

In de volgende delen van deze tekst gaan we wat dieper in op de basisprincipes van enkele belangrijke onderzoeksthema's binnen logisch programmeren. Het zijn enkele van de thema's waarin onze onderzoeksgroep een belangrijke rol speelt.

2 Abstracte Interpretatie

Abstraheren is een basistechniek in het arsenaal van ingenieur en wetenschapper. Het laat hem toe om aspecten die geen of weinig invloed hebben op een bestudeerd fenomeen weg te laten en aldus de complexiteit van het onderzochte systeem drastisch te verminderen. Ook bij het bestuderen van het gedrag van programma's is het gebruik van abstracties essentieel. Stel dat men wil nagaan welke mogelijke waarden een bepaalde veranderlijke kan aannemen op een bepaald punt in het programma, dan zou men het programma

kunnen uitvoeren en nakijken welke waarden de betrokken veranderlijke daar aanneemt. De meeste programma's kunnen echter uitgevoerd worden met een quasi onbeperkt aantal verschillende gegevens. Daarenboven maakt een programma meestal gebruik van iteratie of recursie, en het aantal iteraties kan ook weer nagenoeg onbeperkt zijn. Het is dus in het algemeen onmogelijk om alle mogelijke uitvoeringen te beschouwen, en na te gaan welke waarden onze veranderlijke daarbij aanneemt. De enige uitweg is om een abstracte of symbolische uitvoering van het programma te beschouwen.

Met *abstracte* berekeningen is zowat iedereen vertrouwd. Toen we leerden vermenigvuldigen werd ons ook de negenproef aangeleerd: we maakten een abstractie van de twee operanden, voerden daarop een *abstracte* vermenigvuldiging uit en vergeleken het resultaat met dat van de concrete vermenigvuldiging. Wat later, wanneer we leerden over gehele en reële getallen, maakten we kennis met de tekenregel bij de vermenigvuldiging: een positief getal werd geabstraheerd als "+", een negatief getal als "−", en verder leerden we volgende abstracte vermenigvuldiging:

$$\begin{array}{ccccc} + & * & + & = & + \\ + & * & - & = & - \\ - & * & + & = & - \\ - & * & - & = & + \end{array}$$

Ook een computerprogramma voert elementaire bewerkingen uit, en deze zou men op gelijkaardige manier kunnen abstraheren. Voor elke bewerking een abstracte tegenhanger definiëren die de abstractie van een stel operanden omzet in een abstractie van het resultaat volstaat echter niet. Om een altijd eindigende analyse te bekomen is het essentieel dat elke abstracte operand de abstractie is van een *verzameling* mogelijke concrete operanden. Het is ook noodzakelijk dat er voor elke mogelijke verzameling concrete operanden een abstractie bestaat. Beschouwt men als domein van een operand de verzameling van de gehele of de reële getallen, dan volstaan de abstracties $[+]$ (voor verzamelingen van positieve getallen), $[-]$ (voor verzamelingen van negatieve getallen) en $[0]$ (voor de verzameling $\{0\}$) niet. Immers een verzameling $\{-5, -1, 3\}$ heeft geen abstractie. Het abstracte domein moet minstens uitgebreid worden met een element $[\top]$ dat een abstractie is voor elke verzameling getallen. Noteer dat $[\top]$ ook een abstractie is van de verzameling $\{3, 5, 8\}$, echter $[+]$ is een meer nauwkeurige abstractie. Men zou aan dit abstracte domein ook nog de waarden $[+, 0]$ (abstractie van de niet negatieve getallen) en $[-, 0]$ (abstractie van de niet positieve getallen) kunnen toevoegen en aldus een domein bekomen dat een grotere precisie toelaat. Belangrijker is om het domein ook uit te breiden met $[\bot]$ als abstractie van de lege verzameling. Dit laat o.a. toe om een abstractie te hebben van het resultaat van een falende operatie (vb. delen door 0). De belangrijkste reden om dit element toe te voegen zal verderop duidelijk worden.

In het begin van de jaren 70 werd abstracte interpretatie systematisch bestudeerd door P. en R. Cousot. Een overzicht van de ontwikkelde theorie vindt men in [2]. Zij gebruikten de Galois connectie voor het beschrijven van het verband tussen concreet en abstract domein. In ons hogergenoemd voorbeeld is de toestand van een berekening gekarakteriseerd door de waarde die onze veranderlijke aanneemt. Een element van het concrete domein

bestaat uit een verzameling van toestanden, dus een verzameling van mogelijke waarden. Het concrete domein zelf is dus de verzameling van alle mogelijke verzamelingen van toestanden. Deze verzameling is partieel geordend door de relatie *is deelverzameling van* ($\subseteq$). Bij elke abstracte interpretatie is het concrete domein een partieel geordende verzameling. Een element van het abstracte domein is een *beschrijving* van een element in het concrete domein, het abstracte domein zelf is de verzameling van alle mogelijke beschrijvingen. In ons voorbeeld is het abstracte domein de verzameling $\{[+], [-], [0], [\top], [\bot]\}$. Ook deze verzameling is partieel geordend. De orde volgt uit de betekenis van de beschrijvingen en de orderelatie in het concrete domein. We noteren deze orderelatie als $\sqsubseteq$. Zo is bvb. $[-] \sqsubseteq [\top]$, daar de negatieve getallen een deelverzameling zijn van alle getallen.

De functie die een abstract element afbeeldt op zijn betekenis in het concrete domein wordt concretisatiefunctie genoemd en wordt voorgesteld door γ (vb. $\gamma([+]) = \{1, 2, 3, 4, \ldots\}$). De functie die een element van het concrete domein op zijn (beste) abstractie afbeeldt is de abstractiefunctie en wordt voorgesteld door α (vb. $\alpha(\{2, 3\}) = [+]$). Het is duidelijk dat de orderelaties $\subseteq$ en $\sqsubseteq$ in het abstracte en concrete domein, en de functies α en γ niet onafhankelijk van elkaar kunnen gekozen worden. Het noodzakelijke verband wordt uitgedrukt door de *Galois connectie*:

- α en γ zijn overal gedefinieerd en monotoon.
- voor elk element c van het concrete domein moet gelden: $c \subseteq \gamma(\alpha(c))$. (vb. $\{1, 2\} \subseteq \gamma(\alpha(\{1, 2\})) = \{1, 2, \ldots\}$)
- voor elk element a van het abstract domein moet gelden: $\alpha(\gamma(a)) \sqsubseteq a$ (vb $[+] = \alpha(\gamma([+])) \sqsubseteq [+]$). Indien het abstracte domein geen overbodige elementen bevat (elementen die geen *beste* abstractie zijn van een element uit het concreet domein), zoals in ons voorbeeld, dan geldt zelfs de gelijkheid.

Uit deze voorwaarden volgt o.a. dat elk concreet element een beste abstractie heeft en dat abstractie en concretisatie orde bewarend zijn ($a_1 \sqsubseteq a_2$ impliceert dat $\gamma(a_1) \subseteq \gamma(a_2)$ en $c_1 \subseteq c_2$ impliceert dat $\alpha(c_1) \sqsubseteq \alpha(c_2)$).

De Galois connectie vormt een stevige basis voor het symbolisch uitvoeren van het programma op gegevens uit het abstracte domein. Het enige wat ontbreekt is een abstracte tegenhanger voor elke concrete operatie. Die abstracte tegenhanger moet een veilige benadering berekenen van de resultaten die men bekomt door de concrete operatie toe te passen op alle gegevens die door de abstracte operanden beschreven worden.

Formeel:
Stel dat $f(c_1, \ldots, c_n)$ een operator is die n-operanden gebruikt en $f^A(a_1, \ldots, a_n)$ de abstracte tegenhanger is. $f^A(a_1, \ldots, a_n)$ is veilig indien
$\alpha\{f(c_1, \ldots, c_n) | c_1 \in \gamma(a_1), \ldots, c_n \in \gamma(a_n)\} \sqsubseteq f^A(a_1, \ldots, a_n)$.

Bijvoorbeeld:

$$
\begin{array}{llll|llll}
[+] & *^{\mathcal{A}} & [+] & = [+] & [+] & +^{\mathcal{A}} & [+] & = [+] \\
[+] & *^{\mathcal{A}} & [-] & = [-] & [+] & +^{\mathcal{A}} & [-] & = [\top] \\
[+] & *^{\mathcal{A}} & [0] & = [0] & [+] & +^{\mathcal{A}} & [0] & = [+] \\
[-] & *^{\mathcal{A}} & [+] & = [-] & [-] & +^{\mathcal{A}} & [+] & = [\top] \\
[-] & *^{\mathcal{A}} & [-] & = [+] & [-] & +^{\mathcal{A}} & [-] & = [-] \\
[-] & *^{\mathcal{A}} & [0] & = [0] & [-] & +^{\mathcal{A}} & [0] & = [-] \\
[0] & *^{\mathcal{A}} & [+] & = [0] & [0] & +^{\mathcal{A}} & [+] & = [+] \\
[0] & *^{\mathcal{A}} & [-] & = [0] & [0] & +^{\mathcal{A}} & [-] & = [-] \\
[0] & *^{\mathcal{A}} & [0] & = [0] & [0] & +^{\mathcal{A}} & [0] & = [0]
\end{array}
$$

Dit zijn veilige abstracties van de vermenigvuldiging en van de optelling.

Het enige overblijvende probleem is de behandeling van lussen (iteratie of recursie): de abstracte uitvoering mag slechts een eindig aantal iteraties doorlopen. Dit wordt opgevangen door het raamwerk van abstracte interpretatie en steunt op de monotoniciteit van α en γ. Voor de mogelijke waarden van een veranderlijke in een programmapunt binnen de lus werkt men met een hypothese a (die initieel gelijkgesteld wordt aan $[\bot]$, de abstractie van de lege verzameling). Dan wordt de lus symbolisch uitgevoerd en zo nodig wordt de waarde a vergroot tot een waarde a'. Gebeurt dit, dan wordt de lus nogmaals uitgevoerd, nu met a' als hypothese, gebeurt dit niet, dan is een veilige benadering bereikt. $[\top]$ is steeds een veilige benadering. Door er zorg voor te dragen dat $[\top]$ na een eindig aantal iteraties als hypothese gebruikt wordt, zal men dus steeds in een eindig aantal stappen tot een veilig resultaat komen (met een eindig abstract domein is dit steeds het geval, bij oneindige abstracte domeinen moet hieraan speciale zorg besteed worden).

Binnen een dergelijk raamwerk bestaat het uitwerken van een programma-analyse in het kiezen van een abstract domein en in het ontwikkelen van veilige abstracties voor de primitieve operaties die in de taal aanwezig zijn. Dit blijven niet-triviale taken die nauw met elkaar verbonden zijn. Het domein moet enerzijds eenvoudig genoeg zijn om een analyse met aanvaardbare efficiëntie toe te laten, anderzijds moet het rijk genoeg zijn, niet alleen om de relevante eigenschappen weer te geven, maar ook om de abstracties van de primitieve operaties met de nodige precisie te kunnen definiëren. Binnen een zuivere logische programmeertaal is er in feite maar één primitieve operatie, namelijk unificatie. Ze veilig abstraheren is echter niet eenvoudig. Het effect van een unificatie $X = Y$ waarbij X en Y vrij zijn is dat de twee veranderlijken aan elkaar gebonden worden. Wanneer later bvb. X geünificeerd wordt met a, dan wordt Y eveneens aan a gebonden. Een analyse die met enige precisie eigenschappen over de bindingen van de veranderlijken wil bekomen zal dus op één of andere manier ook in de abstracties moeten bijhouden welke bindingen tussen veranderlijken er mogelijk zijn. Een belangrijk deel van het onderzoek is dan ook gericht op het ontwikkelen van abstracte domeinen, en het onderling vergelijken van hun precisie en efficiëntie. Dit soort onderzoek wordt vergemakkelijkt door de beschikbaarheid van voor logische programmeertalen specifieke raamwerken die in een vroegere faze van het onderzoek ontwikkeld werden. Binnen een dergelijk raamwerk hoeven enkel abstracties ontwikkeld te worden van enkele primitieve operaties, unificatie is daarvan de belangrijkste.

Uiteindelijk is het de bedoeling om de verzamelde informatie ergens voor te gebruiken. Het belangrijkste toepassingsdomein is op het vlak van implementatie. Een logisch programma kan potentieel op veel manieren gebruikt worden. De vertaling moet in alle mogelijkheden voorzien, en slechts tijdens de uitvoering is bekend welk van de mogelijke gevallen geselecteerd moet worden. Abstracte interpretatie kan allerlei informatie verzamelen die toelaat om veel specifiekere en veel efficiëntere code aan te maken. Een tweetal prototype implementaties hebben abstracte interpretatie met sukses in de vertaler geïntegreerd en tonen opmerkelijke snelheidsverbeteringen, in zoverre dat voor sommige programma's dezelfde uitvoeringssnelheid gehaald wordt als voor gelijkaardige programma's in C. Ook bij parallelle implementaties speelt abstracte interpretatie een belangrijke rol. De analyse laat toe om tijdens de vertaling te bepalen of twee taken elkaar al of niet kunnen beïnvloeden en of ze voldoende werk bevatten om ze in twee verschillende processen onder te brengen. Dit laat toe om het extra werk dat een parallelle uitvoering in vergelijking met een sequentiële uitvoering meebrengt, drastisch te reduceren. Ook in heel wat andere domeinen, zoals programma-specialisatie, analyse van het eindigen van de uitvoering van programma's, ... speelt abstracte interpretatie van logische programma's een belangrijke rol. Logisch programmeren is een domein waarbinnen onderzoek rond abstracte interpretatie zeer veel bijgedragen heeft en wellicht het domein waarbinnen concreet bruikbare toepassingen het verst doorgedrongen zijn.

3 Programma-specialisatie

Programma-specialisatie is een vorm van programma-transformatie, waarbij programma's met een te brede functionaliteit worden gespecialiseerd met betrekking tot het enger verband waarin men ze wenst te gebruiken.

De onderliggende motivatie is dat bij de ontwikkeling van nieuwe software het vaak wenselijk is om gebruik te maken van generische, breder toepasbare componenten. Het herbruiken van *bestaande* programma's bij de ontwikkeling van nieuwe toepassingen is hiervan meestal een voorbeeld: die programma's zijn vaak binnen een ander, algemener verband ontwikkeld en zijn niet specifiek gericht op de vereiste functionaliteit in de huidige toepassing. Het hergebruiken van die software biedt toch voordelen op het vlak van het beperken van de ontwikkelingsduur, het beheersen van de complexiteit van de toepassing en het verzekeren van correctheid. Maar ook bij de ontwikkeling van nieuwe programma-code is het vaak wenselijk om de programma's algemener en meer generisch te ontwerpen dan strikt nodig. Dit verbetert de aanpasbaarheid en de herbruikbaarheid voor toekomstige toepassingen.

Een dergelijke aanpak heeft naast de vermelde voordelen ook één uitgesproken nadeel: het gebruik van te algemene oplossingen leidt meestal tot verlies aan uitvoeringssnelheid. Er is dan ook nood aan methodes die te algemene programma's kunnen omzetten naar meer specifieke programma's. Bij voorkeur zijn dergelijke methodes volledig geautomatiseerd, opdat de bovenvermelde voordelen van het gebruik van algemenere programma's (zoals korte ontwikkelingsduur en correctheid) niet in het gedrang zouden komen. Automatische

programma-specialisatie biedt hier een oplossing.

Ter illustratie, veronderstel dat we te maken hebben met een toepassing waarin het nodig is matrices van welbepaalde dimensies te transponeren. Meer bepaald, stel dat het transponeren van zowel $1 \times n$-matrices, met n een willekeurige dimensie, als 2×2-matrices vereist is. Een algemeen programma voor transpositie, voor willekeurige dimensies, lost alle vereiste taken hierrond op. Dit wordt verwezenlijkt door het volgende logische programma:

```
transp(Mat,[]):-
        nilrij(Mat).
transp(Mat,[Tr_rij|Tr_rest]):-
        maakrij(Mat,Tr_rij,Rest),
        transp(Rest,Tr_rest).

nilrij([]).
nilrij([[]|Rest]):-
        nilrij(Rest).

maakrij([],[],[]).
maakrij([[Eerste|Restrij]|Rijen],[Eerste|Tr_rij],
                                  [Restrij|Restrijen]):-
        maakrij(Rijen,Tr_rij,Restrijen).
```

In dit programma wordt een matrix gerepresenteerd als een lijst van lijsten. Opeenvolgende deellijsten beschrijven de opeenvolgende rijen in de matrix. Het predicaat *maakrij* drukt uit dat de eerste rij van de getransponeerde matrix de elementen bevat uit de eerste kolom van de gegeven matrix. Tegelijk drukt *maakrij* het verband uit tussen de gegeven matrix en de matrix die ontstaat door de eerste kolom uit de gegeven matrix te verwijderen. Het predicaat *nilrij* drukt uit hoe, gebruik makende van de lijst-representatie, alle rijen van de getransponeerde matrix dienen te worden afgesloten met de lege lijst [].

Het programma *transp* is duidelijk te algemeen in het licht van de veronderstellingen rond de beoogde toepassing. De specifiekere toepassing waarin we eigenlijk geïnteresseerd zijn is het transponeren van 2×2-matrices, uitgedrukt als oproepen van de vorm:

```
?-transp([[X,Y],[U,V]],Trans).
```

en het transponeren van $1 \times n$-matrices, oproepen van het type:

```
?-trans([Rij],Trans).
```

De twee hierboven vermelde vraagstellingen beschrijven eigenlijk elk een patroon van verschillende vraagstellingen waarin we geïnteresseerd zijn. Meer bepaald wensen we dat het programma dat we ontwikkelen alle mogelijke instantiaties (van het juiste type) van deze vraagstellingen kan oplossen. Met instantiaties bedoelen we hier die vraagstellingen die ontstaan door één of meerdere veranderlijken in de vraagstelling te vervangen door termen (van het beoogde type). Een specifieker programma dat alle mogelijke instantiaties van de vraagstelling

```
?- transp_1([[X,Y],[U,V]],Trans).
```

correct beantwoordt, bestaat slechts uit één enkel feit:

```
transp_1([[X,Y],[U,V]],[[X,U],[Y,V]]).
```

Analoog is een specifieker programma dat alle mogelijke instantiaties van

```
?-transp_2([Rij],Trans).
```

correct beantwoordt:

```
transp_2([[]],[]).
transp_2([[H|T]],[[H]|S]):-
        transp_2([T],S).
```

Programma-specialisatie methodes leiden de nieuwe progamma's *transp_1* en *transp_2* automatisch af uit het gegeven programma *transp* en de beoogde vraagstellingspatronen. Daarenboven vervangen deze specialisatie-methodes elke predicaatoproep in de toepassing door de gepaste equivalente oproep naar *transp_1* of *transp_2*. Meer nog, de beoogde vraagstellingspatronen hoeven zelfs niet aan de programma-specialisator te worden gespecifieerd. In de meeste gevallen kunnen die patronen automatisch worden afgeleid. Aldus wordt een "best of both worlds" gerealiseerd: de voordelen van het (her-)gebruik van generische of algemene programma's, samen met efficiënte programma-code.

Teneinde de bruikbaarheid van deze technieken verder te verduidelijken bespreken we een tweede voorbeeld: de specialisatie van een vertolker. In logische programmeertalen worden de meer geavanceerde toepassingen vaak ontwikkeld met behulp van zogenaamde *meta-vertolkers*. Een meta-vertolker voor taal L is een programma geschreven in de taal L dat

- programma's geschreven in - mogelijk een extensie van - L uitvoert,
- mogelijk extra functionaliteiten, buiten het strikte uitvoeren van de programma's in (die uitbreiding van) L, toevoegt.

Een typisch voorbeeld hiervan is de ontwikkeling van expert-systeem-kernen. Het volgende meta-programma biedt de essentie van een zeer eenvoudige expert-systeem-kern voor kennisbanken:

```
los_op([],[]).
los_op([Doel|Rest],[Bewijs_Doel|Bewijs_Rest]):-
        los_1_op(Doel,Bewijs_Doel),
        los_op(Rest,Bewijs_Rest).

los_1_op(Doel,bewezen(Doel <- Bewijs_Cond)):-
        regel(Doel if Cond),
```

```
        los_op(Cond,Bewijs_Cond).
los_1_op(Doel,gebruiker(Doel)):-
        naar_gebruiker(Doel),
        vraag_gebruiker(Doel).
```

In dit geval ondersteunt de meta-vertolker geen uitbreidingen aan de taal. Hij ondersteunt wel twee uitbreidingen aan de functionaliteit:

- het registreren van het bewijs dat tijdens de berekening voor een bepaalde vraagstelling wordt opgebouwd (het tweede argument van *los_op* en *los_1_op*),
- de ondersteuning van ongedefinieerde predicaten die aan de gebruiker ter beantwoording dienen te worden voorgelegd (de tweede regel voor *los_1_op*).

De manier waarop het geregistreerde bewijs later wordt benut, teneinde de gebruiker uitleg te verschaffen over de bekomen resultaten, wordt in het voorbeeld niet gespecifieerd. Ook de definitie van het predikaat *vraag_gebruiker* zullen we hier niet verder uitwerken. Het programma is alleen zinvol indien er daarnaast ook een stel regels wordt opgegeven waarrond bepaalde vraagstellingen dienen te worden opgelost. Zo'n stel regels beschrijven bijvoorbeeld een familie-relatie kennisbank, waarvan de volgende regels deel uitmaken:

```
regel(grootmoeder(X,Y) if [moeder(X,Z), ouder(Z,Y)]).
regel(ouder(X,Y) if [moeder(X,Y)]).
regel(ouder(X,Y) if [vader(X,Y)]).
```

Daarenboven kunnen we het programma uitbreiden met de specificatie van welke kennisbank-relaties door de gebruiker moeten worden aangevuld. Bijvoorbeeld:

```
naar_gebruiker(moeder(X,Y)).
naar_gebruiker(vader(X,Y)).
```

Dit programma is algemener dan strikt nodig, door het feit dat de meta-vertolker (de predicaten *los_op* en *los_1_op*) toepasbaar zijn op om het even welke kennisbank. Ze zijn niet specifiek voor de beschouwde familie-relatie regels. Daarnaast is het best mogelijk dat we alleen geïnteresseerd zijn in welbepaalde patronen van vraagstellingen rond de familie-kennisbank. Zo kunnen we bijvoorbeeld mogelijk alleen geïnteresseerd zijn in vragen van het type

```
?- los_op(grootmoeder(X,Y),Bewijs).
```

of instantiaties hiervan.

Voor dit vraagstellingspatroon genereert programma-specialisatie het volgende meer specifieke programma:

```
los_op_gr(X,Y,bewezen(grootmoeder(X,Y) <-
 [Bewijs_moeder,Bewijs_ouder])):-
        los_op_moe(X,Z,Bewijs_moeder),
```

```
        los_op_oud(Z,Y,Bewijs_ouder).

los_op_oud(X,Y,bewezen(ouder(X,Y) <- Bewijs_moe)):-
        los_op_moe(X,Y,Bewijs_moe).
los_op_oud(X,Y,bewezen(ouder(X,Y) <- Bewijs_va)):-
        los_op_va(X,Y,Bewijs_va).

los_op_moe(X,Y,gebruiker(moeder(X,Y))):-
        vraag_gebruiker(moeder(X,Y)).

los_op_va(X,Y,gebruiker(vader(X,Y))):-
        vraag_gebruiker(vader(X,Y)).
```

Dit programma geeft dezelfde antwoorden op een vraagstelling

```
?- los_op_gr(X,Y,Bewijs).
```

als *los_op* op het overeenkomstige vraagstellingspatroon. Alleen gedraagt het gespecialiseerde programma zich niet meer als een vertolker. De oorspronkelijke regelbank werd door middel van specialisatie vertaald met betrekking tot de gegeven vertolker. Het nieuwe programma kan rechtstreeks uitgevoerd worden, zonder nood aan een bijkomende laag van vertolking. Ook het testen of een predicaat al dan niet aan de gebruiker moet worden voorgelegd is in de gespecialiseerde versie niet meer nodig. Tenslotte is het programma ook nog specifiek gemaakt voor de klasse van vraagstellingen waarin we geïnteresseerd waren (bevraging van de *grootmoeder* relatie). Deze drie optimalisaties resulteren voor dit soort van specialisatie-voorbeelden meestal in een efficiëntie winst van een factor 10 of meer.

Een gedetailleerde bespreking van de methodes die kunnen aangewend worden om deze specialisaties automatisch te genereren zou ons in deze uiteenzetting te ver voeren. De elementaire transformatie-stap waarop alle technieken gebaseerd zijn is het *ontvouwen* van predicaat-oproepen. Ontvouwen is een transformatie waarbij één enkele stap uit de berekening wordt toegepast op het beschouwde vraagstellingspatroon. Daarbij wordt een nieuwe definitie voor het predicaat gegenereerd die zowel het vraagstellingspatroon als de oude definitie van het predicaat in rekening brengt. Beschouw ter illustratie opnieuw het vraagstellingspatroon

```
?- transp([[X,Y],[U,V]],Trans).
```

voor het algemene *transp* programma. Eén enkele ontvouwing van dit patroon met behulp van de beide regels voor *transp* geeft aanleiding tot twee specifiekere regels:

```
transp([[X,Y],[U,V]],[]):-
        nilrij ([[X,Y],[U,V]]).
transp([[X,Y],[U,V],[Tr_rij|Tr_Rest]):-
        maakrij([[X,Y],[U,V]],Tr_rij,Rest),
        transp (Rest,Tr_rest).
```

Het verder toepassen van ontvouw-transformaties op de rechterleden van deze regels resulteert uiteindelijk in het programma *transp_1*.

De belangrijkste problemen hierbij situeren zich op het vlak van de controle van deze elementaire transformatie. Wanneer dient het openvouwen te worden beëindigd? Hier spelen, naast het voor de hand liggende criterium van terminatie, ook een aantal criteria omtrent de kwaliteit van de gegenereerde specialisaties een belangrijke rol.

Een ander controle aspect betreft de generatie van recursieve gespecialiseerde regels. Alleen dank zij zeer verfijnde controle technieken slagen de specialisatie-methodes erin het ontvouw-proces zo te sturen, dat de gespecialiseerde regels zichzelf op een correcte manier kunnen oproepen.

4 Implementatie van Prolog

De meest verspreide logische programmeertaal is Prolog. Om Prolog te implementeren, kunnen we vertrekken van een implementatie van een imperatieve taal zoals C en dan de zaken die eigen zijn aan Prolog en niet in C bestaan, gradueel toevoegen in de implementatie. Dat is niet de gewone manier van doen: in 1983 heeft D. H. D. Warren [5] een beschrijving van een abstracte machine voor Prolog gepubliceerd die het vertrekpunt is geworden voor elke succesvolle hedendaagse Prologimplementatie (zie ook [1]). De beschrijving van die abstracte machine - WAM genoemd - is verwarrend, omdat het niet altijd duidelijk is welke delen thuishoren in de categorie van *optimalisaties*, en welke in de categorie *noodzakelijk* voor *Prolog*. Daarom onze aanpak vertrekkend van een taal zoals C.

We bekijken eerst een deelverzameling van Prolog, mini-Prolog, die benaderd kan worden door een subset van C, mini-C, namelijk Prologprogramma's waarin elk predicaat slechts één Horn-uitdrukking heeft. Dan is mini- C de verzameling van C-programma's zonder globale veranderlijken, met in het lichaam van de functies enkel functie-oproepen en met hoogstens één toekenning aan elke veranderlijke.

De enige controle-structuur, nodig om mini-Prolog uit te voeren, is, net zoals voor mini-C, een uitvoeringsstapel, waarop bij elke procedure-oproep een "stackframe" wordt gezet, met daarin een wijzer naar het vorige stackframe (prevF) en het terugkeeradres (CP). Figuur 1 laat de toestand van de stapel zien voor het programma

```
main :- b, c.
b :- d, e.
```

en de vraagstelling

```
?- main.
```

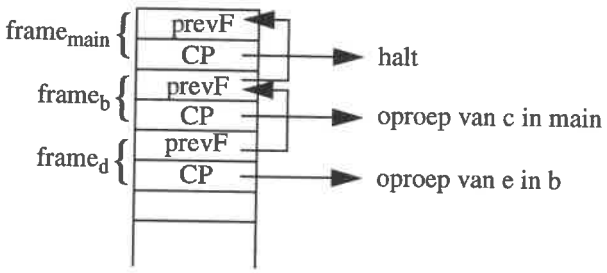
dat correspondeert met het mini-C-programma

```
main() { b() ; c() ; }
b() { d() ; e() ; }
```

net nadat het predicaat **d** is opgeroepen. Vertalen van bijvoorbeeld de procedure main, levert de volgende intuïtief duidelijke code op:

```
allocate
call b
call c
return
```

waarbij de conventie is dat de opgeroepen procedure zijn stackframe aanmaakt.



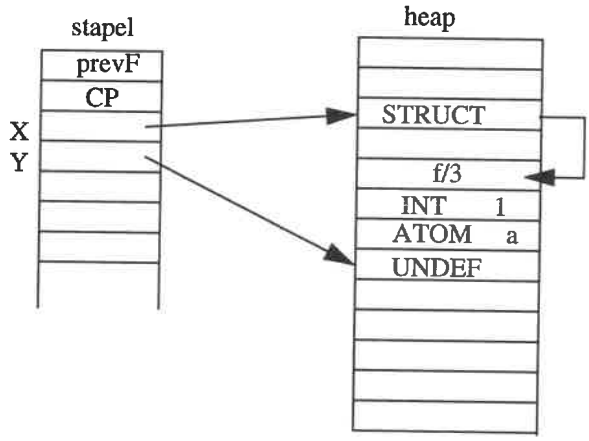
Figuur 1: Stapel na oproep van **d**.

Mini-Prolog laat ook veranderlijken toe: in mini-C corresponderen die met lokale veranderlijken waarvoor plaats gemaakt wordt in het stackframe van de procedure; we doen net hetzelfde voor mini-Prolog: elke veranderlijke krijgt een vaste plaats in het stackframe. Van een Prolog-veranderlijke is het in het algemeen niet mogelijk om het *type* te bepalen tijdens de vertaling; daarom volstaat het niet om voor een veranderlijke waaraan een *toekenning* gebeurt bij te houden welke waarde zij heeft, we moeten ook een merkteken *tag* hebben die zegt van welk type die waarde is. We onderscheiden hier voor de eenvoud slechts enkele tags: de tag **ATOM** - gebruikt voor waarden die atomaire namen zijn - en **INT** - voor gehele getallen. Een derde tag **STRUCT** hebben we nodig om gestructureerde objecten (zoals de term **f(1,a)**) voor te stellen. Bovendien drukken gestructureerde termen ons met de neus op het feit dat niet alleen het type van een veranderlijke pas gekend is tijdens de uitvoering, maar ook de hoeveelheid geheugen nodig om haar waarde voor te stellen. Het is dus niet mogelijk om in een stackframe voldoende plaats te voorzien voor elke waarde die een veranderlijke tijdens de uitvoering zou kunnen aannemen. Daarom voeren we een opslagplaats in - de *heap* - waar we alle waarden bewaren. In de stackframes worden enkel verwijzingen naar die waarden bijgehouden. Bovendien hebben we een voorstelling van vrije veranderlijken (in C zouden we zeggen *niet geïnitialiseerde* veranderlijken) nodig. We gebruiken hiervoor de tag **UNDEF**. Schematisch geeft Figuur 2 weer hoe de stapel en de heap eruit zien tijdens de uitvoering van de Horn-uitdrukking

```
main :- X = f(1,a,Y) , b(X,Y) .
```

net voordat **b** opgeroepen wordt.

Het corresponderende mini-C programma ziet er met een beetje fantasie uit als:



Figuur 2: Stapel met heap

```
main()
  {Y = init(UNDEF,0);
   X = init(STRUCT,"f",3,init(INT,1),init(ATOM,"a"),init(var,Y));
   b(X,Y);
  }
```

Om parameters door te geven, gebruiken we de afspraak die ook bestaat op sommige RISC-machines: er is een vast stel registers - in de WAM argumentenregisters genaamd - waarin de oproeper de argumenten klaarzet, t.t.z. verwijzingen naar waarden die op de heap staan. Elke oproep hergebruikt hetzelfde stel registers.

Nu kunnen we unificatie bekijken; laat ons dat doen op het eenvoudige voorbeeld:

```
main :- X = f(a,A), Y = f(B,b), X = Y.
```

De eerste twee voorkomens van $=/2$ in deze Horn-uitdrukking zijn geen unificaties: het is immers de eerste keer dat X (en Y) voorkomen en we beschouwen dit als een toekenning. In $X = Y$ zijn beide veranderlijken niet meer vrij en we hebben unificatie nodig. Figuur 3 laat de toestand van stapel en heap zien voor en na unificatie. De unificatie slaagt en sommige UNDEF's zijn nu *geïnitialiseerd* en overschreven met hun waarde.

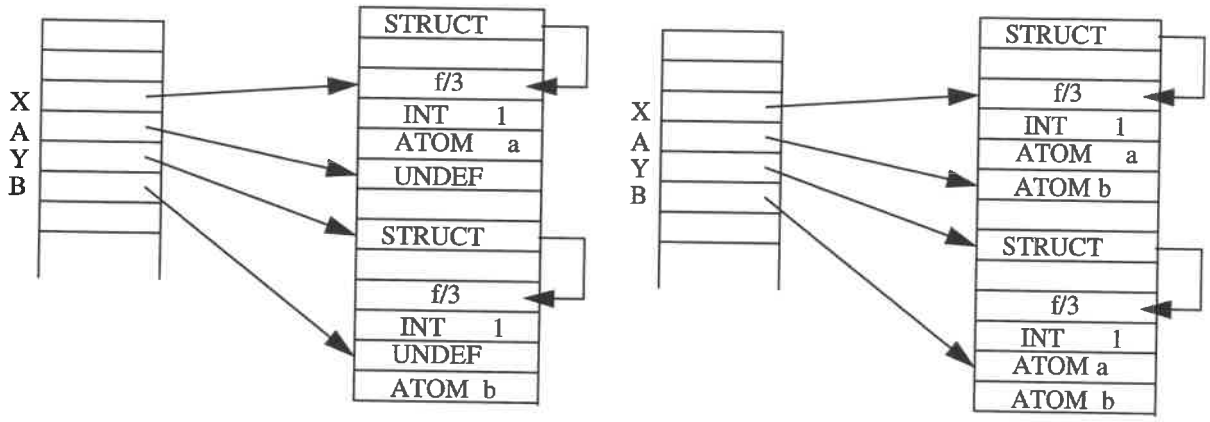
Het is dezelfde unificatie die gebruikt wordt in een elementaire resolutiestap. Immers, de hoofding

```
a(f(1,A)) :- ...
```

kan herschreven worden tot

```
a(<Areg1>) :- <Areg1> = f(1,A) , ...
```

waarbij <Areg1> niet een Prologveranderlijke voorstelt, maar het object waar argumentenregister 1 naar verwijst. Als we een unificatiefunctie *unify* gegeven veronderstellen, dan ziet de overeenstemmende code voor de Horn-uitdrukking



Figuur 3: Effect van een unificatie

```
a(X,g(A,R)) :- a(X,R).
```

eruit als:

```
allocate 4 % er zijn 3 lokale veranderlijken
           % + 1 hulpveranderlijke
Hulp = g(A,R)
unify(<Areg2>,Hulp)
move R, <Areg2>
call a/2
return
```

Bovenstaande ingrediënten zijn voldoende voor de uitvoering van mini-Prolog en de afwijking met een naïve implementatie van mini-C bestaat in de heap.

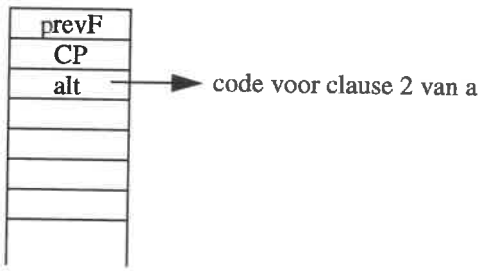
WAM is een geoptimaliseerde versie van het bovenstaande, in zoverre dat WAM last-call-optimalisatie toepast (een veralgemening van staart-recursie-optimalisatie), en voor termen die tijdens de vertaling gekend zijn vermijdt om de algemene *unify* op te roepen, t.t.z. de oproep wordt vervangen door een instructie die gespecialiseerd is voor het soort term. Verder vermijdt de WAM het toewijzen van een stackframe voor Horn-uitdrukkingen met slechts één of geen enkel doel en specialiseert WAM een aantal instructies waarbij lijststructuren betrokken zijn.

Mini-Prolog is weinig krachtig: in het bijzonder missen we een *if-then-else* constructie. Prolog heeft die in de vorm van meerdere Horn-uitdrukkingen voor een predicaat, waarbij de tweede uitdrukking geprobeerd wordt indien de eerste al zijn resultaten heeft afgeleverd, dan de derde, en zo verder, met het mechanisme dat bekend staat als *backtracking*. We hebben geen equivalent voor backtracking in imperatieve talen. Toch kunnen we op een machineniveau backtracking beschrijven als een variante van voorwaartse uitvoering, als volgt: we breiden elk stackframe uit met een veld *alt* dat aangeeft welke de volgende (alternatieve) Horn-uitdrukking is. Schematisch voor het predicaat:

```
a(X,Y) :- b(Y,X).
```

```
a(A,B) :- c(C).
```

ziet het stackframe van de activatie van de eerste Horn-uitdrukking van a/2 er nu uit als in Figuur 4.



Figuur 4: Stackframe met voorziening voor backtracking

Als een unificatie faalt, gaat het uitvoeringsmechanisme verder met de code waarnaar het alt-veld van het jongste stackframe wijst en - om bij een later falen de daaropvolgende Horn-uitdrukking te kunnen proberen - wordt het alt-veld gezet op het adres van de daaropvolgende uitdrukking. Bij het proberen van de laatste Horn-uitdrukking wordt er in het alt-veld een aanduiding gezet dat er geen alternatieven meer zijn: bij falen zal het stackframe verwijderd worden en een ander stackframe, waarvoor alternatieven zijn, zal op de top van de stapel komen. Schematisch kan men dus de code voor een predicaat met n alternatieven beschrijven als:

```
entry:  allocate m, alt2 % m = aantal lokale veranderlijken, alt2
                         %het adres van de tweede Horn-uitdrukking
        <instructies voor hoofding en lichaam van eerste uitdrukking>
        return
alt2:   allocate m, alt3
        <instructies voor hoofding en lichaam van tweede uitdrukking>
        return
alt3:   ...
        ...
altn:   allocate m, altn+1
        <instructies voor hoofding en lichaam van laatste uitdrukking>
        return
altn+1: verwijder top stackframe
        backtrack naar alt van top stackframe
```

Wij zijn nog een beetje vaag over wat backtrack juist wil zeggen, maar we beschrijven eerst kort wat er nog ontbreekt aan dit mechanisme van backtracking: als de machine backtrackt, moet de toestand van de machine hersteld worden; dit wil zeggen: alle machineregisters moeten teruggezet worden naar de waarden die ze hadden op het ogenblik dat het jongste stackframe gecreëerd werd. Dat kan enkel door in het stackframe naast het

alternatief ook alle machineregisters te bewaren, in het bijzonder wijzers naar de top van de heap en de stapel, het CP register en de argumenten die bij de oproep horen. Verder moet ook de toestand van de heap hersteld worden, t.t.z. alle bindingen die door unificatie zijn gedaan moeten ongedaan gemaakt worden. Unificatie overschrijft soms een UNDEF op de heap met een waarde en we kunnen deze wijzigingen bijhouden op een spoorstapel (*trail* genoemd in de WAM) die de adressen bevat van de UNDEF-velden die door unificatie veranderd werden. Bij het backtracken zal dan het topgedeelte van de spoorstapel gebruikt worden om diezelfde velden terug op UNDEF te zetten. In het stackframe komt dus ook de *top-of-trail* wijzer te staan.

Het verschil tussen bovenstaande beschrijving van een Prolog-implementatie en WAM zit in het feit dat WAM een stackframe in twee splitst: het ene gedeelte dient voor backtracking en wordt het keuzepunt genoemd, het andere gedeelte dient voor de voorwaartse uitvoering en wordt de omgeving genoemd. Optimalisaties op beide afzonderlijk worden dan ingevoerd, zoals bijvoorbeeld het vroeg verwijderen van keuzepunten of het vermijden van keuzepunten voor deterministische oproepen. WAM definieert een hele reeks gespecialiseerde instructies die deze optimalisaties implementeren.

De WAM heeft model gestaan voor bijna alle efficiënte hedendaagse Prolog-implementaties en ook voor talen die een (soms parallelle) variante zijn van Prolog. Ook ProLog_by_BIM - een commerciële Prolog die ontstond uit een samenwerkingsproject tussen het departement computerwetenschappen van de K.U.Leuven en BIM - gebruikt in grote mate de WAM. Het onderzoek van onze groep in verband met implementatie heeft zich beziggehouden met bijna alle aspecten van de WAM: registertoekenning (het hergebruik van argumentenregisters die dikwijls gerealiseerd worden als hardware registers), indexatie (verband houdend met optimalisaties voor de keuzepunten), unificatie (waarbij een optimale specialisatie van de unify-functie werd ontworpen), dynamische code (dit is code die door "assert" toegevoegd wordt tijdens de uitvoering en waarvoor de WAM geen beschrijving geeft), geheugenrecuperatie (in het bijzonder van de heap en dit zowel voor sequentiële Prolog als voor een OR-parallel systeem) en mechanismen voor "error-recovery". Verder is er een parallelle implementatie gemaakt van AKL, een Prolog-achtige taal die doelen niet van links naar rechts uitvoert in een lichaam, maar volgens het principe van deterministische doelen eerst; AKL laat bovendien zowel AND- als OR-parallellisme toe.

5 Besluit

Het onderzoek over computationale logica heeft geleid tot een technologie die met sukses kan ingezet worden bij de ontwikkeling van complexe programmatuur. Heden worden de mogelijkheden die deze technologie biedt te weinig door de industrie benut. De toenemende vraag naar steeds complexere en terzelfder tijd meer betrouwbare programmatuur maakt dat talen en technieken die een hoger niveau aan abstractie bieden aan belang zullen winnen. Men kan dus vermoeden dat declaratieve technieken, o.a. zoals ontwikkelt binnen computationele logica, in de toekomst aan belang zullen winnen.

Referenties

[1] H. Aït-Kaci. *Warren's Abstract Machine, A tutorial reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991.

[2] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[3] R. A. Kowalski. *Logic for Problem Solving.* Elsevier/North Holland, Amsterdam, 1979.

[4] J. W. Lloyd. *Foundations of Logic Programming.* Springer Series : Symbolic Computation - Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.

[5] D. H. D. Warren. An abstract Prolog instruction set. Technical report, SRI International, Artificial Intelligence Center, 1983.